GDB Manual

The GNU Source-Level Debugger

First Edition, GDB version 1.11

December 1986

Richard M. Stallman

Copyright © 1986 Richard M. Stallman.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "Distribution" and "GDB General Public License" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the sections entitled "Distribution" and "GDB General Public License" may be included in a translation approved by the author instead of in the original English.

TOWER

GDB Manual

The GNU Source-Level Debugger

First Edition, GDB version 1.11

December 1986

Richard M. Stallman

Copyright © 1986 Richard M. Stallman.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "Distribution" and "GDB General Public License" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the sections entitled "Distribution" and "GDB General Public License" may be included in a translation approved by the author instead of in the original English.

Table of Contents

Summary of GDB 1

GDB General Public License 3
Copying Policies 3
NO WARRANTY 5

1. GDB Input Conventions 7

2. Specifying GDB's Files 9
2.1 Specifying Files with Arguments 9
2.2 Specifying Files with Commands 9

3. Switches and Arguments for GDB 11
3.1 Mode Switches 11
3.2 File-specifying Switches 11
3.3 Other Arguments 11

4. Running Your Program Under GDB 13
4.1 Your Program's Arguments 13
4.2 Your Program's Environment 14
4.3 Your Program's Working Directory 14

5. Stopping and Continuing 17
5.1 Signals 17
5.2 Breakpoints 18
5.2.1 Setting Breakpoints 19
5.2.2 Clearing Breakpoints 20
5.2.3 Disabling Breakpoints 21
5.2.4 Break Conditions 22
5.2.5 Commands Executed on Breaking 23
5.2.6 "Cannot Insert Breakpoints" Error 25
5.3 Continuing 25
5.4 Stepping 26

6. Examining the Stack 29
6.1 Stack Frames 29
6.2 Backtraces 30
6.3 Selecting a Frame 30
6.4 Information on a Frame 31

7. Examining Source Files 33
7.1 Printing Source Lines 33
7.2 Specifying Source Directories 34

8. Examining Data 37
8.1 Expressions 37
8.2 Program Variables 38
8.3 Artificial Arrays 38
8.4 Formats 39
8.4.1 Examining Memory 40
8.5 Automatic Display 42
8.6 Value History 43
8.7 Convenience Variables 44
8.8 Registers 45
8.8.1 Examples 45

9. Examining the Symbol Table 47

10. Altering Execution 49
10.1 Assignment to Variables 49
10.2 Continuing at a Different Address 49
10.3 Giving the Program a Signal 50
10.4 Returning from a Function 50

11. Canned Sequences of Commands 53
11.1 User-Defined Commands 53
11.2 Command Files 53
11.3 Commands for Controlled Output 54

Command Index 55

Concept Index 57

Summary of GDB

The purpose of a debugger such as GDB is to allow you to execute another program while examining what is going on inside it. We call the other program "your program" or "the program being debugged".

GDB can do four kinds of things (plus other things in support of these):

1. Start the program, specifying anything that might affect its behavior.
2. Make the program stop on specified conditions.
3. Examine what has happened, when the program has stopped, so that you can see bugs happen.
4. Change things in the program, so you can correct the effects of one bug and go on to learn about another without having to recompile first.

GDB General Public License

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share GDB. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of GDB, that you receive source code or else can get it if you want it, that you can change GDB or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of GDB, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for GDB. If GDB is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Fundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change GDB.

Copying Policies

1. You may copy and distribute verbatim copies of GDB source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice "Copyright © 1986 Free Software Foundation, Inc." (or with the year updated if that is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the GDB program a copy of this License Agreement along with the program.
2. You may modify your copy or copies of GDB source code or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
    1. cause the modified files to carry prominent notices stating who last changed such files and the date of any change; and

2. cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GDB or any part thereof, to be freely distributed and licensed to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to third parties, at your option).
3. if the modified program serves as a debugger, cause it when started running in the simplest and usual way, to print an announcement including a valid copyright notice "Copyright © 1986 Free Software Foundation, Inc." (or with the year updated if appropriate), saying that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License Agreement.

3. You may copy and distribute GDB or any portion of it in compiled, executable or object code form under the terms of Paragraphs 1 and 2 above provided that you do the following:
   1. cause each such copy of GDB to be accompanied by the corresponding machine-readable source code; or
   2. cause each such copy of GDB to be accompanied by a written offer, with no time limit, to give any third party free (except for a nominal shipping charge) machine readable copy of the corresponding source code; or
   3. in the case of a recipient of GDB in compiled, executable or object code form (without the corresponding source code) you shall cause copies you distribute to be accompanied by a copy of the written offer of source code which you received along with the copy of GDB.
4. You may not copy, sublicense, distribute or transfer GDB except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GDB is void and your rights to use GDB under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

NO WARRANTY

BECAUSE GDB IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GDB "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE GDB PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL FREE SOFTWARE FOUNDATION, INC., RICHARD M. STALLMAN, AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GDB AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH PROGRAMS NOT DISTRIBUTED BY FREE SOFTWARE FOUNDATION, INC.) THE PROGRAM, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

1. GDB Input Conventions

GDB is invoked with the shell command 'gdb'. Once started, it reads commands from the terminal until you tell it to exit.

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. Some command names do not allow arguments.

GDB command names may always be abbreviated if the abbreviation is unambiguous. Sometimes even ambiguous abbreviations are allowed; for example, 's' is specially defined as equivalent to 'step' even though there are other commands whose names start with 's'. Possible command abbreviations are often stated in the documentation of the individual commands.

A blank line as input to GDB means to repeat the previous command verbatim. Certain commands do not allow themselves to be repeated this way; these are commands for which unintentional repetition might cause trouble and which you are unlikely to want to repeat. Certain others ('list' and 'x') act differently when repeated because that is more useful.

A line of input starting with '#' is a comment; it does nothing. This is useful mainly in command files (See section 11.2 [Command Files], page 53).

To exit GDB, use the 'quit' command (abbreviated 'q'). Ctrl-c will not exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It is safe to type Ctrl-c at any time because GDB does not allow it to take effect until a time when it is safe.

2. Specifying GDB's Files

GDB needs to know the filename of the program to be debugged. To debug a core dump of a previous run, GDB must be told the filename of the core dump.

2.1 Specifying Files with Arguments

The usual way to specify the executable and core dump file names is with two command arguments given when you start GDB. The first argument is used as the file for execution and symbols, and the second argument (if any) is used as the core dump file name. Thus,

```
gdb progm core
```

specifies 'progm' as the executable program and 'core' as a core dump file to examine. (You do not need to have a core dump file if what you plan to do is debug the program interactively.)

See chapter 3 [Switches], page 11, for full information on command switches and arguments for GDB.

2.2 Specifying Files with Commands

Usually you specify the files for GDB to work with by giving arguments when you invoke GDB. But occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify the files you want to use. In these situations the GDB commands to specify new files are useful.

exec-file *filename*
: Specify that the program to be run is found in *filename*. If you do not specify a directory and the file is not found in GDB's working directory, GDB will use the environment variable 'PATH' as a list of directories to search, just as the shell does when looking for a program to run.

symbol-file *filename*
: Read symbol table information from file *filename*. 'PATH' is searched when necessary. Most of the time you will use both the 'exec-file' and 'symbol-file' commands on the same file.

'`symbol-file`' with no argument clears out GDB's symbol table.

core-file *filename*

Specify the whereabouts of a core dump file to be used as the "contents of memory". Note that the core dump contains only the writable parts of memory; the read-only parts must come from the executable file.

'`core-file`' with no argument specifies that no core file is to be used.

kill Cancel running the program under GDB. This could be used if you wish to debug a core dump instead. GDB ignores any core dump file if it is actually running the program, so the '`kill`' command is the only sure way to go back to using the core dump file.

info files Print the names of the executable and core dump files currently in use by GDB, and the file from which symbols were loaded.

While all three file-specifying commands allow both absolute and relative file names as arguments, GDB always converts the file name to an absolute one and remembers it that way.

The '`symbol-file`' command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

3. Switches and Arguments for GDB

When you invoke GDB, you can pass commands telling it what files to operate on and what other things to do.

3.1 Mode Switches

'-nx' Do not execute commands from the file '`./.gdbinit`'. Normally, the commands in this file are executed (if the file exists) after all the command switches and arguments have been processed.

'-q' "Quiet". Do not print the usual introductory messages.

'-batch' Run in batch mode. Exit with code 1 after processing all the command files specified with '-x' (and '`./.gdbinit`', if not inhibited). Exit also if, due to an error, GDB would otherwise attempt to read a command from the terminal.

3.2 File-specifying Switches

All the switches and command line arguments given are processed in sequential order. The order makes a difference when the '-x' command is used.

'-s *file*' Read symbol table from file *file*.

'-e *file*' Use file *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

'-se *file*' Read symbol table from file *file* and use it as the executable file.

'-c *file*' Use file *file* as a core dump to examine.

'-x *file*' Execute GDB commands from file *file*.

3.3 Other Arguments

If there are arguments to GDB that are not switches or associated with switches, the first one specifies the symbol table and executable file name (as if it were preceded by '-se') and the second one specifies a core dump file name (as if it were preceded by '-c').

4. Running Your Program Under GDB

To start your program under GDB, use the 'run' command. The program must already have been specified using the 'exec-file' command or with an argument to GDB (see chapter 2 [Files], page 9); what 'run' does is create an inferior process, load the program into it, and set it in motion.

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify them, which you must do *before* starting the program. (You can change them after starting the program, but such changes do not affect the program unless you start it over again.)

The *arguments.*

You specify the arguments to give the program as the arguments of the 'run' command.

The *environment.*

The program normally inherits its environment from GDB, but you can use the GDB commands 'set-environment' and 'unset-environment' to change parts of the environment that will be given to the program.

The *working directory.*

The program inherits its working directory from GDB. You can set GDB's working directory with the 'cd' command in GDB.

After the 'run' command, the debugger does nothing but wait for your program to stop. See chapter 5 [Stopping], page 17.

4.1 Your Program's Arguments

You specify the arguments to give the program as the arguments of the 'run' command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to the program.

'run' with no arguments uses the same arguments used by the previous 'run'.

The command 'set-args' can be used to specify the arguments to be used the next time the program is run. If 'set-args' has no arguments, it means to use no arguments the next time the program is run. If you have run your program with arguments and want to run it again with no arguments, this is the only way to do so.

4.2 Your Program's Environment

The *environment* consists of a set of *environment variables* and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inerited by all the other programs you run. When debugging, it can be useful to try running the program with different environments without having to start the debugger over again.

info environment *varname*
: Print the value of environment variable *varname* to be given to your program when it is started. This command can be abbreviated '`i env` *varname*'.

info environment
: Print the names and values of all environment variables to be given to your program when it is started. This command can be abbreviated '`i env`'.

set-environment *varname* = *value*
: Sets environment variable *varname* to *value*, for your program only, not for GDB itself. *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. This command can be abbreviated as short as '`set-e`'.

unset-environment *varname*
: Remove variable *varname* from the environment to be passed to your program. This is different from '`set-env` *varname* =' because '`unset-environment`' makes a variable not be defined at all, which is distinguishable from an empty value. This command can be abbreviated '`unset`'.

In 4.2bsd on the Vax, the shell has a bug that crashes it when it is invoked by GDB before it can run your program. This bug happens if the total size of all environment variables is above a certain threshold. If this bug bites you, you can evade it by unsetting some environment variables.

4.3 Your Program's Working Directory

Each time you start your program with 'run', it inherits its working directory from the current working directory of GDB. GDB's working directory is initially whatever it inherited from its superior, but you can specify the working directory for GDB with the '`cd`' command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See chapter 2 [Files], page 9.

cd *directory*
: Set GDB's working directory to *directory*.

pwd
: Print GDB's working directory.

5. Stopping and Continuing

When you run a program normally, it runs until exiting. The purpose of using a debugger is so that you can stop it before that point; or so that if the program runs into trouble you can find out why.

5.1 Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, SIGINT is the signal a program gets when you type `Ctrl-c`; SIGSEGV is the signal a program gets from referencing a place in memory far away from all the areas in use; SIGALRM occurs when the alarm clock timer goes off (which happens only if the program has requested an alarm).

Some signals, including SIGALRM, are a normal part of the functioning of the program. Others, such as SIGSEGV, indicate errors; these signals are *fatal* (kill the program immediately) if the program has not specified in advance some other way to handle the signal. SIGINT does not indicate an error in the program, but it is normally fatal so it can carry out the purpose of `Ctrl-c`: to kill the program.

GDB has the ability to detect any occurrence of a signal in the program running under GDB's control. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like SIGALRM (so as not to interfere with their role in the functioning of the program) but to stop the program immediately whenever an error signal happens. You can change these settings with the 'handle' command. You must specify which signal you are talking about with its number.

info signal Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

handle *signalnum keywords...*

Change the way GDB handles signal *signalnum*. The *keywords* say what change to make.

To use the 'handle' command you must know the code number of the signal you are concerned with. To find the code number, type 'info signal' which prints a table of signal names and numbers.

The keywords allowed by the handle command can be abbreviated. Their full names are

stop	GDB should stop the program when this signal happens. This implies the 'print' keyword as well.
print	GDB should print a message when this signal happens.
nostop	GDB should not stop the program when this signal happens. It may still print a message telling you that the signal has come in.
noprint	GDB should not mention the occurrence of the signal at all. This implies the 'nostop' keyword as well.
pass	GDB should allow the program to see this signal; the program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.
nopass	GDB should not allow the program to see this signal.

When a signal has been set to stop the program, the program cannot see the signal until you continue. It will see the signal then, if 'pass' is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the 'handle' command with 'pass' or 'nopass' to control whether that signal will be seen by the program when you later continue it.

You can also use the 'signal' command to prevent the program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. See section 10.3 [Signaling], page 50.

5.2 Breakpoints

A *breakpoint* makes your program stop whenever a certain point in the program is reached. You set breakpoints explicitly with GDB commands, specifying the place where the program should stop by line number, function name or exact address in the program. You can add various other conditions to control whether the program will stop.

Each breakpoint is assigned a number when it is created; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on the program until you enable it again.

The command 'info break' prints a list of all breakpoints set and not cleared, showing their numbers, where in the program they are, and any special features in use for them. Disabled breakpoints are included in the list, but marked as disabled. 'info break' with a breakpoint

number as argument lists only that breakpoint. The convenience variable '$_' and the default examining-address for the 'x' command are set to the address of the last breakpoint listed (see section 8.4.1 [Memory], page 40).

The default examine address for the 'x' command is changed to the starting address of the line, so that 'x/i' is sufficient to begin examining the machine code (see section 8.4.1 [Memory], page 40). Also, this address is saved as the value of the convenience variable '$_' (see <undefined> [Convenience], page <undefined>).

5.2.1 Setting Breakpoints

Breakpoints are set with the 'break' command (abbreviated 'b'). You have several ways to say where the breakpoint should go.

break *function*
: Set a breakpoint at entry to function *function*.

break *linenum*
: Set a breakpoint at line *linenum* in the current source file. That file is the last file whose source text was printed. This breakpoint will stop the program just before it executes any of the code on that line.

break *filename:linenum*
: Set a breakpoint at line *linenum* in source file *filename*.

break *filename:function*
: Set a breakpint at entry to function *function* found in file *filename*. Specifying a filename as well as a function name is superfluous except when multiple files contain similarly named functions.

break **address*
: Set a breakpoint at address *address*. You can use this to set breakpoints in parts of the program which do not have debugging information or source files.

break
: Set a breakpoint at the next instruction to be executed in the selected stack frame (see chapter 6 [Stack], page 29). This is a silly thing to do in the innermost stack frame because the program would stop immediately after being started, but it is very useful with another stack frame, because it will cause the program to stop as soon as control returns to that frame.

break ... if *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero. '...' stands for one of the

possible arguments described above (or no argument) specifying where to break. See section 5.2.4 [Conditions], page 22, for more information on breakpoint conditions.

tbreak *args*

Set a breakpoint enabled only for one stop. *args* are the same as in the 'break' command, and the breakpoint is set in the same way, but the breakpoint is automatically *disabled* the first time it is hit.

There is no problem with setting more than one breakpoint at the same place, but this only becomes useful when the breakpoints are conditional (see section 5.2.4 [Conditions], page 22).

5.2.2 Clearing Breakpoints

It is often necessary to eliminate a breakpoint once it has done its job and you no longer want the program to stop there. This is called *clearing* or 'deleting' the breakpoint. A breakpoint that has been cleared no longer exists in any sense.

With the 'clear' command you can clear breakpoints according to where they are in the program. With the 'delete' command you can clear individual breakpoints by specifying their breakpoint numbers.

It is not necessary to clear a breakpoint to proceed past it. GDB automatically ignores breakpoints in the first instruction to be executed when you continue execution at the same address where the program stopped.

clear

Clear any breakpoints at the next instruction to be executed in the selected stack frame (see section 6.3 [Selection], page 30). When the innermost frame is selected, this is a good way to clear a breakpoint that the program just stopped at.

clear *function*

clear *filename:function*

Clear any breakpoints set at entry to the function *function*.

clear *linenum*

clear *filename:linenum*

Clear any breakpoints set at or within the code of the specified line.

delete *bnums...*

Delete the breakpoints of the numbers specified as arguments. A breakpoint deleted is forgotten completely.

5.2.3 Disabling Breakpoints

Rather than clearing a breakpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been cleared, but remembers the information on the breakpoint so that you can *enable* it again later.

You disable and enable breakpoints with the '`enable`' and '`disable`' commands, specifying one or more breakpoint numbers as arguments. Use '`info break`' to print a list of breakpoints if you don't know which breakpoint numbers to use.

A breakpoint can have any of four different states of enablement:

- Enabled. The breakpoint will stop the program. A breakpoint made with the '`break`' command starts out in this state.
- Disabled. The breakpoint has no effect on the program.
- Enabled once. The breakpoint will stop the program, but when it does so it will become disabled. A breakpoint made with the '`tbreak`' command starts out in this state.
- Enabled for deletion. The breakpoint will stop the program, but immediately after it does so it will be deleted permanently.

You change the state of enablement of a breakpoint with the following commands:

disable *bnums...*

Disable the specified breakpoints. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later.

enable *bnums...*

Enable the specified breakpoints. They become effective once again in stopping the program, until you specify otherwise.

enable once *bnums...*

Enable the specified breakpoints temporarily. Each will be disabled again the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

enable delete *bnums...*

Enable the specified breakpoints to work once and then die. Each of the breakpoints will be deleted the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

Aside from the automatic disablement or deletion of a breakpoint when it stops the program, which happens only in certain states, the state of enablement of a breakpoint changes only when one of the commands above is used.

5.2.4 Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a boolean expression in your programming language. A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

Break conditions may have side effects, and may even call functions in your program. These may sound like strange things to do, but their effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop the program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see section 5.2.5 [Break Commands], page 23).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the '`break`' command. See section 5.2.1 [Set Breaks], page 19. They can also be changed at any time with the '`condition`' command:

condition *bnum expression*

> Specify *expression* as the break condition for breakpoint number *bnum*. From now on, this breakpoint will stop the program only if the value of *expression* is true (nonzero, in C). *expression* is not evaluated at the time the '`condition`' command is given.

condition *bnum*

> Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

A special feature is provided for one kind of condition: to prevent the breakpoint from doing anything until it has been reached a certain number of times. This is done with the *ignore count* of the breakpoint. When the program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues.

ignore *bnum count*

> Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, it will not stop.

To make the breakpoint stop the next time it is reached, specify a count of zero.

cont *count* Continue execution of the program, setting the ignore count of the breakpoint that the program stopped at to *count*. Continuing through the breakpoint does not itself count as one of *count*. This command is allowed only when the program stopped due to a breakpoint.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, the condition will start to be checked.

Note that you could achieve the effect of the ignore count with a condition such as '`$foo-- <= 0`' using a debugger convenience variable that is decremented each time. That is why the ignore count is considered a special case of a condition. See section 8.7 [Convenience Vars], page 44.

5.2.5 Commands Executed on Breaking

You can give any breakpoint a series of commands to execute when the program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

commands *bnum*

Specify commands for breakpoint number *bnum*. The commands themselves appear on the following lines. Type a line containing just '`end`' to terminate the commands.

To remove all commands from a breakpoint, use the command '`commands`' and follow it immediately by '`end`'; that is, give no commands.

It is possible for breakpoint commands to start the program up again. Simply use the '`cont`' command, or '`step`', or any other command to resume execution. However, any remaining breakpoint commands are ignored. When the program stops again, GDB will act according to why that stop took place.

If the first command specified is '`silent`', the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If the remaining commands too print nothing, you will see no sign that the breakpoint was reached at all. '`silent`' is not really a command; it is meaningful only at the beginning of the commands for a breakpoint.

The commands '`echo`' and '`output`' that allow you to print precisely controlled output are often useful in silent breakpoints. See section 11.3 [Output], page 54.

For example, here is how you could use breakpoint commands to print the value of x at entry to foo whenever it is positive. We assume that the newly created breakpoint is number 4; 'break' will print the number that is assigned.

```
break foo if x>0
commands 4
silent
echo x is\040
output x
echo \n
cont
end
```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the 'cont' command so that the program does not stop, and start with the 'silent' command so that no output is produced. Here is an example:

```
break 403
commands 5
silent
set x = y + 4
cont
end
```

One deficiency in the operation of automatically continuing breakpoints under Unix appears when your program uses raw mode for the terminal. GDB switches back to its own terminal modes (not raw) before executing commands, and then must switch back to raw mode when your program is continued. This causes any pending terminal input to be lost.

In the GNU system, this will be fixed by changing the behavior of terminal modes.

Under Unix, when you have this problem, you might be able to get around it by putting your actions into the breakpoint condition instead of commands. For example

```
condition 5  (x = y + 4), 0
```

is a condition expression that will change x as needed, then always have the value 0 so the program will not stop. Loss of input is avoided here because break conditions are evaluated without changing the terminal modes. When you want to have nontrivial conditions for performing the side effects,

the operators '&&', '||' and '? ... :' may be useful.

5.2.6 "Cannot Insert Breakpoints" Error

Under Unix, breakpoints cannot be used in a program if any other process is running that program. Attempting to run or continue the program with a breakpoint in this case will cause GDB to stop it.

When this happens, you have two ways to proceed:

1. Remove or disable the breakpoints, then continue.
2. Suspend GDB, and copy the file containing the program to a new name. Resume GDB and use the 'exec-file' command to specify that GDB should run the program under that name. Then start the program again.

5.3 Continuing

After your program stops, most likely you will want it to run some more if the bug you are looking for has not happened yet.

cont Continue running the program at the place where it stopped.

If the program stopped at a breakpoint, the place to continue running is the address of the breakpoint. You might expect that continuing would just stop at the same breakpoint immediately. In fact, 'cont' takes special care to prevent that from happening. You do not need to clear the breakpoint to proceed through it after stopping at it.

You can, however, specify an ignore-count for the breakpoint that the program stopped at, by means of an argument to the 'cont' command. See section 5.2.4 [Conditions], page 22.

If the program stopped because of a signal other than SIGINT or SIGTRAP, continuing will cause the program to see that signal. You may not want this to happen. For example, if the program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but the program would probably terminate immediately as a result of the fatal signal once it sees the signal. To prevent this, you can continue with 'signal 0'. See section 10.3 [Signaling], page 50. You can also act in

advance to prevent the program from seeing certain kinds of signals, using the 'handle' command (see section 5.1 [Signals], page 17).

5.4 Stepping

Stepping means setting your program in motion for a limited time, so that control will return automatically to the debugger after one line of code or one machine instruction. Breakpoints are active during stepping and the program will stop for them even if it has not gone as far as the stepping command specifies.

step
: Proceed the program until control reaches a different line, then stop it and return to the debugger. This command is abbreviated 's'.

step *count*
: Proceed as in 'step', but do so *count* times. If a breakpoint or a signal not related to stepping is reached before *count* steps, stepping stops right away.

next
: Similar to 'step', but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the 'next' command was given. This command is abbreviated 'n'.

  An argument is a repeat count, as in 'step'.

finish
: Continue running until just after the selected stack frame returns. The program will stop sooner if a breakpoint is hit or a fatal signal takes place.

  Contrast this with the 'return' command (see section 10.4 [Returning], page 50).

stepi
si
: Proceed one machine instruction, then stop and return to the debugger.

  It is often useful to do 'display/i $pc' when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop. See section 8.5 [Auto Display], page 42.

  An argument is a repeat count, as in 'step'.

nexti
ni
: Proceed one machine instruction, but if it is a subroutine call, proceed until the subroutine returns.

  An argument is a repeat count, as in 'next'.

A typical techique for using stepping is to put a breakpoint (see section 5.2 [Breakpoints], page 18) at the beginning of the function or the section of the program in which a problem is believed to lie, and then step through the suspect area, examining the variables that are interesting, until the problem happens.

The 'cont' command can be used after stepping to resume execution until the next breakpoint or signal.

6. Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, the information about where in the program the call was made from is saved in a block of data called a *stack frame*. The frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in the program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in.

When the program stops, GDB automatically selects the currently executing frame and describes it briefly as the '`frame`' command does (see section 6.4 [Frame Info], page 31).

6.1 Stack Frames

The call stack is divided up into contiguous pieces called *frames*; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function main. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing

one of those bytes whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are to give you a way of talking about stack frames in GDB commands.

Many GDB commands refer implicitly to one stack frame. GDB records a stack frame that is called the *selected* stack frame; you can select any frame using one set of GDB commands, and then other commands will operate on that frame. When your program stops, GDB automatically selects the innermost frame.

6.2 Backtraces

A backtrace is a summary of how the program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

backtrace

bt — Print a backtrace of the entire stack: one line per frame for all frames in the stack.

You can stop the backtrace at any time by typing the system interrupt character, normally `Control-C`.

backtrace *n*

bt *n* — Similar, but stop after *n* frames.

Each line in a backtrace shows the frame number, the program counter, the function and its arguments, and the source file name and line number (if known). The program counter may be omitted if it as it the beginning of the source line. This is the same as the first of the two lines printed when you select a frame.

6.3 Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

frame *n* Select frame number *n*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is main's frame.

frame *addr*

Select the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when the program has multiple stacks and switches between them.

up *n* Select the frame *n* frames up from the frame previously selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

down *n* Select the frame *n* frames down from the frame previously selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one.

All of these commands end by printing some information on the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

```
#3  main (argc=3, argv=??, env=??) at main.c, line 67
67          read_input_file (argv[i]);
```

After such a printout, the '`list`' command with no arguments will print ten lines centered on the point of execution in the frame. See section 7.1 [List], page 33.

6.4 Information on a Frame

There are several other commands to print information about the selected stack frame.

frame This command prints a brief description of the selected stack frame. It can be abbreviated '`f`'. With an argument, this command is used to select a stack frame; with no argument, it does not change which frame is selected, but still prints the same information.

info frame This command prints a verbose description of the selected stack frame, including the address of the frame, the addresses of the next frame in (called by this frame) and the next frame out (caller of this frame), the address of the frame's arguments, the program counter saved in it (the address of execution in the caller frame), and which

registers were saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command.

info args Print the arguments of the selected frame, each on a separate line.

info locals Print the local variables of the selected frame, each on a separate line. These are all variables declared static or automatic within all program blocks that execution in this frame is currently inside of.

7. Examining Source Files

GDB knows which source files your program was compiled from, and can print parts of their text. When your program stops, GDB spontaneously prints the line it stopped in. Likewise, when you select a stack frame (see section 6.3 [Selection], page 30), GDB prints the line which execution in that frame has stopped in. You can also print parts of source files by explicit command.

7.1 Printing Source Lines

To print lines from a source file, use the '`list`' command (abbreviated '`l`'). There are several ways to specify what part of the file you want to print.

Here are the forms of '`list`' command most commonly used:

list *linenum*
: Print ten lines centered around line number *linenum* in the current source file.

list *function*
: Print ten lines centered around the beginning of function *function*.

list
: Print ten more lines. If the last lines printed were printed with a '`list`' command, this prints ten lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see chapter 6 [Stack], page 29), this prints ten lines centered around that line.

list –
: Print ten lines just before the lines last printed.

Repeating a '`list`' command with RET discards the argument, so it is equivalent to typing just '`list`'. This is more useful than listing the same lines again. An exception is made for an argument of '`-`'; that argument is preserved in repetition so that each repetition moves up in the file.

In general, the '`list`' command expects you to supply zero, one or two *linespecs*. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for '`list`':

list *linespec*
: Print ten lines centered around the line specified by *linespec*.

list *first,last*
: Print lines from *first* to *last*. Both arguments are linespecs.

list -*last* Print ten lines ending with *last*.

list *first*. Print ten lines starting with *first*.

list + Print ten lines just after the lines last printed.

list − Print ten lines just before the lines last printed.

list As described in the preceding table.

Here are the ways of specifying a single source line—all the kinds of linespec.

linenum Specifies line *linenum* of the current source file. When a '`list`' command has two linespecs, this refers to the same source file as the first linespec.

+*offset* Specifies the line *offset* lines after the last line printed. When used as the second linespec in a '`list`' command that has two, this specifies the line *offset* lines down from the first linespec.

−*offset* Specifies the line *offset* lines before the last line printed.

filename:*linenum*
Specifies line *linenum* in the source file *filename*.

function Specifies the line of the open-brace that begins the body of the function *function*.

filename:*function*
Specifies the line of the open-brace that begins the body of the function *function* in the file *filename*. The file name is needed with a function name only for disambiguation of identically named functions in different source files.

**address* Specifies the line containing the program address *address*. *address* may be any expression.

One other command is used to map source lines to program addresses.

info line *linenum*
Print the starting and ending addresses of the compiled code for source line *linenum*. The default examine address for the '`x`' command is changed to the starting address of the line, so that '`x/i`' is sufficient to begin examining the machine code (see section 8.4.1 [Memory], page 40). Also, this address is saved as the value of the convenience variable '`$_`' (see <undefined> [Convenience], page <undefined>).

7.2 Specifying Source Directories

Executable programs do not record the directories of the source files they were compiled from, just the names. GDB remembers a list of directories to search for source files; this is called the

source path. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name.

When you start GDB, its source path contains just the current working directory. To add other directories, use the '`directory`' command. **Note that the search path for executable files and the working directory are *not* used for finding source files.**

directory *dirname*
: Add directory *dirname* to the end of the source path.

directory
: Reset the source path to just the current working directory of GDB. This requires confirmation.

  '`directory`' with no argument can cause source files previously found by GDB to be found in a different directory. To make this work correctly, this command also clears out the tables GDB maintains about the source files it has already found.

info directories
: Print the source path: show which directories it contains.

Because the '`directory`' command adds to the end of the source path, it does not affect any file that GDB has already found. If the source path contains directories that you do not want, and these directories contain misleading files with names matching your source files, the way to correct the situation is as follows:

1. Choose the directory you want at the beginning of the source path. Use the '`cd`' command to make that the current working directory.
2. Use '`directory`' with no argument to reset the source path to just that directory.
3. Use '`directory`' with suitable arguments to add any other directories you want in the source path.

8. Examining Data

The usual way of examining data in your program is with the 'print' command (abbreviated 'p'). It evaluates and prints the value of any valid expression of the language the program is written in (for now, C). You type

```
print exp
```

where *exp* is any valid expression, and the value of *exp* is printed in a format appropriate to its data type.

A more low-level way of examining data is with the 'x' command. It examines data in memory at a specified address and prints it in a specified format.

8.1 Expressions

Many different GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is legal in an expression in GDB. This includes conditional expressions, function calls, casts and string constants.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into pointer so as to examine a structure at that address in memory.

GDB supports three kinds of operator in addition to those of programming languages:

@ '@' is a binary operator for treating parts of memory as arrays. See section 8.3 [Arrays], page 38, for more information.

:: ':' allows you to specify a variable in terms of the file or function it is defined in. See section 8.2 [Variables], page 38.

{*type*} *addr*
Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around nonunary operators, just as in a cast). This construct is allowed regardless of what kind of data is officially supposed to reside at *addr*.

8.2 Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see section 6.3 [Selection], page 30); they must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

```
foo (a)
     int a;
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

the variable a is usable whenever the program is executing within the function foo, but the variable b is visible only while the program is executing inside the block in which b is declared.

8.3 Artificial Arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

This can be done by constructing an *artificial array* with the binary operator '@'. The left operand of '@' should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes memory immediately following those that hold the first element, and so on. Here is an example. If a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of array with

```
p *array@len
```

The left operand of '@' must reside in memory. Array values made with '@' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. (It would probably appear in an expression via the value history, after you had printed it out.)

8.4 Formats

GDB normally prints all values according to their data types. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or an instruction. These things can be done with *output formats.*

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the 'print' command with a slash and a format letter. The format letters supported are:

'x' Regard the bits of the value as an integer, and print the integer in hexadecimal.

'd' Print as integer in signed decimal.

'u' Print as integer in unsigned decimal.

'o' Print as integer in octal.

'a' Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

'c' Regard as an integer and print it as a character constant.

'f' Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see section 8.8 [Registers], page 45), type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the 'print' command with just a format and no expression. For example, 'p/x' reprints the last value in hex.

8.4.1 Examining Memory

The command 'x' (for 'examine') can be used to examine memory under explicit control of formats, without reference to the program's data types.

'x' is followed by a slash and an output format specification, followed by an expression for an address. The expression need not have a pointer value (though it may); it is used as an integer, as the address of a byte of memory.

The output format in this case specifies both how big a unit of memory to examine and how to print the contents of that unit. It is done with one or two of the following letters:

These letters specify just the size of unit to examine:

'b' Examine individual bytes.

'h' Examine halfwords (two bytes each).

'w' Examine words (four bytes each).

Many assemblers and cpu designers still use 'word' for a 16-bit quantity, as a holdover from specific predecessor machines of the 1970's that really did use two-byte words. But more generally the term 'word' has always referred to the size of quantity that a machine normally operates on and stores in its registers. This is 32 bits for all the machines that GNU runs on.

'g' Examine giant words (8 bytes).

These letters specify just the way to print the contents:

'x' Print as integers in unsigned hexadecimal.

'd' Print as integers in signed decimal.

'u' Print as integers in unsigned decimal.

'o' Print as integers in unsigned octal.

'a' Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

'c' Print as character constants.

'f' Print as floating point. This works only with sizes 'w' and 'g'.

's' Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is however many bytes it takes to reach a null character (including the null character).

'i' Print a machine instruction in assembler syntax (or nearly). The specified unit size is ignored; the number of bytes in an instruction varies depending on the type of machine, the opcode and the addressing modes used.

If either the manner of printing or the size of unit fails to be specified, the default is to use the same one that was used last. If you don't want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is just after the last unit examined. This is why string and instruction formats actually compute a unit-size based on the data: so that the next string or instruction examined will start in the right place. The 'print' command sometimes sets the default address for the 'x' command; when the value printed resides in memory, the default is set to examine the same location. 'info line' also sets the default for 'x', to the address of the start of the machine code for the specified line and 'info breakpoints' sets it to the address of the last breakpoint listed.

When you use RET to repeat an 'x' command, it does not repeat exactly the same: the address specified previously (if any) is ignored, so that the repeated command examines the successive locations in memory rather than the same ones.

You can examine several consecutive units of memory with one command by writing a repeat-count after the slash (before the format letters, if any). The repeat count must be a decimal integer. It has the same effect as repeating the 'x' command that many times except that the output may be more compact with several units per line.

```
x/10i $pc
```

Prints ten instructions starting with the one to be executed next in the selected frame. After doing this, you could print another ten following instructions with

```
x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the 'x' command are not put in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables '$_' and '$__'.

After an 'x' command, the last address examined is available for use in expressions in the convenience variable '$_'. The contents of that address, as examined, are available in the convenience variable '$__'.

If the 'x' command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were printed on the last line of output.

8.5 Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB will print its value each time the program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

showing item numbers, expressions and their current values.

display *exp*
: Add the expression *exp* to the list of expressions to display each time the program stops.

display/*fmt* *exp*
: For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arranges to display it each time in the specified format *fmt*,

display/*fmt* *addr*
: For *fmt* 'i' or 's', or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time the program stops. Examining means in effect doing 'x/*fmt* *addr*'. See section 8.4.1 [Memory], page 40.

undisplay *n*
: Remove item number *n* from the list of expressions to display.

display
: Display the current values of the expressions on the list, just as is done when the program stops.

info display
: Print the list of expressions to display automatically, each one with its item number, but without showing the values.

8.6 Value History

Every value printed by the 'print' command is saved for the entire session in GDB's *value history* so that you can refer to it in other expressions.

The values printed are given *history numbers* for you to refer to them by. These are successive integers starting with 1. 'print' shows you the history number assigned to a value by printing '$*n* = ' before the value; here *n* is the history number.

To refer to any previous value, use '$' followed by the value's history number. The output printed by 'print' is designed to remind you of this. Just '$' refers to the most recent value in the history, and '$$' refers to the value before that.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component 'next' points to the next one, you can print the contents of the next one with

```
p *$.next
```

It might be useful to repeat this command many times by typing RET.

Note that the history records values, not expressions. If the value of x is 4 and you type

```
print x
set x=5
```

then the value recorded in the value history by the 'print' command remains 4 even though x's value has changed.

info history
: Print the last ten values in the value history, with their item numbers. This is like 'p $$9' repeated ten times, except that 'info history' does not change the history.

info history *n*
: Print ten history values centered on history item number *n*.

8.7 Convenience Variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no effect on further execution of your program. That's why you can use them freely.

Convenience variables have names starting with '$'. Any name starting with '$' can be used for a convenience variable, unless it is one of the predefined set of register names (see section 8.8 [Registers], page 45).

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. Example:

```
set $foo = *object_ptr
```

would save in '$foo' the value contained in the object pointed to by object_ptr.

Using a convenience variable for the first time creates it; but its value is void until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. The convenience variable as an expression has whatever type its current value has.

info convenience

Print a list of convenience variables used so far, and their values. Abbreviated 'i con'.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example:

```
set $i = 0
print bar[$i++]->contents
```
...repeat that command by typing RET.

Some convenience variables are created automatically by GDB and given values likely to be useful.

'$_' The variable '$_' is automatically set by the 'x' command to the last address examined (see section 8.4.1 [Memory], page 40). Other commands which provide a default address for 'x' to examine also set '$_' to that address; these commands include 'info line' and 'info breakpoint'.

'$__' The variable '$__' is automatically set by the 'x' command to the value found in the last address examined.

8.8 Registers

Machine register contents can be referred to in expressions as variables with names starting with '$'. The names of registers are different for each machine; use 'info registers' to see the names used on your machine. The names '$pc' and '$sp' are used on all machines for the program counter register and the stack pointer. Often '$fp' is used for a register that contains a pointer to the current stack frame.

Register values are always integers. There may someday be a way to extract floating values stored in registers.

Register values are relative to the selected stack frame (see section 6.3 [Selection], page 30). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame (with 'frame 0').

Some registers are never saved (typically those numbered zero or one) because they are used for returning function values; for these registers, relativization makes no difference.

info registers

Print the names and relativized values of all registers.

8.8.1 Examples

You could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer with

```
set $sp += 4
```

The last is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected. Setting '`$sp`' is not allowed when other stack frames are selected.

9. Examining the Symbol Table

The commands described in this section allow you to make inquiries for information about the symbols (names of variables, functions and types) defined in your program. This information is found by GDB in the symbol table loaded by the '`symbol-file`' command; it is inherent in the text of your program and does not change as the program executes.

whatis *exp* Print the data type of expression *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place.

whatis Print the data type of '`$`', the last value in the value history.

ptype *typename*

Print a desciption of data type *typename*. *typename* may be the name of a type, or for C code it may have the form '`struct` *struct-tag*', '`union` *union-tag*' or '`enum` *enum-tag*'.

info sources

Print the names of all source files in the program for which there is debugging information.

info functions

Print the names and data types of all defined functions.

info functions *regexp*

Print the names and data types of all defined functions whose names contain a match for regular expression *regexp*. Thus, '`info fun step`' finds all functions whose names include '`step`'; '`info fun ^step`' finds those whose names start with '`step`'.

info variables

Print the names and data types of all variables that are declared outside of functions.

info variables *regexp*

Print the names and data types of all variables, declared outside of functions, whose names contain a match for regular expression *regexp*.

info types Print all data types that are defined in the program.

info types *regexp*

Print all data types that are defined in the program whose names contain a match for regular expression *regexp*.

printsyms *filename*

Write a complete dump of the debugger's symbol data into the file *filename*.

10. Altering Execution

There are several ways to alter the execution of your program with GDB commands.

10.1 Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. For example,

```
print x=4
```

would store the value 4 into the variable x, and then print the value of the assignment expression (which is 4).

If you are not interested in seeing the value of the assignment, use the 'set' command instead of the 'print' command. 'set' is really the same as 'print' except that the expresion's value is not printed and is not put in the value history (see section 8.6 [Value History], page 43). The expression is evaluated only for side effects.

GDB allows more implicit conversions in assignments than C does; you can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that is the same length or shorter.

In C, all the other assignment operators such as '+=' and '++' are supported as well.

To store into arbitrary places in memory, use the '{...}' construct to generate a value of specified type at a specified address (see section 8.1 [Expressions], page 37). For example,

```
set {int}0x83040 = 4
```

10.2 Continuing at a Different Address

jump *linenum*

Resume execution at line number *linenum*. Execution may stop immediately if there is a breakpoint there.

The 'jump' command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line *linenum* is in a different function from the one currently executing, the results may be wild if the two functions expect different patterns of arguments or of local variables. However, even wild results are predictable based on changing the program counter.

jump **address*

Resume execution at the instruction at address *address*.

A similar effect can be obtained by storing a new value into the register '$pc', but not exactly the same.

```
set $pc = 0x485
```

specifies the address at which execution will resume, but does not resume execution. That does not happen until you use the 'cont' command or a stepping command (see section 5.4 [Stepping], page 26).

10.3 Giving the Program a Signal

signal *signalnum*

Resume execution where the program stopped, but give it immediately the signal number *signalnum*.

Alternatively, if *signalnum* is zero, continue execution and give no signal. This may be useful when the program has received a signal and the 'cont' command would allow the program to see that signal.

10.4 Returning from a Function

You can make any function call return immediately, using the 'return' command.

First select the stack frame that you wish to return from (see section 6.3 [Selection], page 30). Then type the 'return' command. If you wish to specify the value to be returned, give that as an argument.

This pops the selected stack frame (and any other frames inside of it), leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the

registers used for returning values of functions.

The '`return`' command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. Contrast this with the '`finish`' command (see section 5.4 [Stepping], page 26), which resumes execution *until* the selected stack frame returns naturally.

11. Canned Sequences of Commands

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

11.1 User-Defined Commands

A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. This is done with the 'define' command.

define *commandname*

Define a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

The definition of the command is made up of other GDB command lines, which are given following the 'define' command. The end of these commands is marked by a line containing 'end'.

document *commandname*

Give documentation to the user-defined command *commandname*. The command *commandname* must already be defined. This command reads lines of documentation just as 'define' reads the lines of the command definition. After the 'document' command is finished, 'help' on command *commandname* will print the documentation you have specified.

You may use the 'document' command again to change the documentation of a command. Redefining the command with 'define' does not change the documentation.

User-defined commands do not take arguments. When they are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

Commands that would ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in user-defined command.

11.2 Command Files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting

with '#') may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When GDB starts, it automatically executes its *init file*, a command file named '`.gdbinit`' in the working directory. You can also request the execution of a command file with the '`source`' command.

source *filename*
: Execute the command file *filename*.

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a command file.

11.3 Commands for Controlled Output

During the execution of a command file or a user-defined command, the only output that appears is what is explicitly printed by the commands of the definition. This section describes two commands useful for generating exactly the output you want.

echo *text*
: Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as '\n' to print a newline. **No newline will be printed unless you specify one.**

  A backslash at the end of *text* is ignored. It is useful for outputting a string ending in spaces, since trailing spaces are trimmed from all arguments. A backslash at the beginning preserves leading spaces in the same way, because '\ ' as an escape sequence stands for a space. Thus, to print ' and `foo` = ', do

```
echo \ and foo = \
```

output *expression*
: Print the value of *expression* and nothing but that value: no newlines, no '$*nn* = '. The value is not entered in the value history either.

output/*fmt* *expression*
: Print the value of *expression* in format *fmt*. See section 8.4 [Formats], page 39, for more information.

Command Index

$

$_ 18, 19, 34, 41
$__ 41

B

break 19

C

cd 15
clear 20
condition 22
core-file 10

D

define 53
delete 20
directory 35
disable 21
display 42
document 53
down 31

E

echo 54
enable 21
exec-file 9

F

finish 26
frame 31

H

handle 17

I

ignore 22
info args 32
info break 18
info convenience 44
info directories 35
info display 42
info environment 14
info files 10
info frame 31
info functions 47
info history 43
info line 34
info locals 32
info registers 45
info signal 17
info sources 47
info types 47
info variables 47

J

jump 49

K

kill 10

L

list 33

N

next 26
nexti 26
ni 26

O

output 54

P

print 37
printsyms 47
ptype 47
pwd 15

Q

quit 7

R

return 50
run 13

S

set 49
set-args 13
set-environment 14
si 26
signal 50
silent 23
source 54
step 26
stepi 26
symbol-file 9

T

tbreak 20

U

undisplay 42
unset-environment 14
up 31

W

whatis 47

X

x 40

Concept Index

$

$ 43
$$ 43

.

.gdbinit 54

A

arguments (to your program) 13
artificial array 38
assignment 49

B

breakpoint commands 23
breakpoints 18

C

call stack 29
clear breakpoint 20
command files 53
conditions 22
convenience variables 44
core dump file 9

D

delete breakpoints 20
directories for source files 34
disabled breakpoints 21

E

enabled breakpoints 21
environment (of your program) 14
examining data 37
examining memory 40
executable file 9
exiting GDB 7
expressions 37

F

formatted output 39
frame 29
frame number 30
frame pointer 29

I

ignore count (of breakpoint) 22
init file 54
initial frame 29
innermost frame 29

O

outermost frame 29
output formats 39

P

printing data 37

R

registers 45
returning from a function 50
running 13

S

selected frame 30
setting variables 49
source path 34
stepping 26
symbol table 9

U

user-defined commands 53

V

value history 43

W

word 40
working directory (of your program) 14